Methods and Models for Combinatorial Optimization, Lab Exercise Part 1

Alberto Nicoletti

A.Y. 2023/2024, University of Padova

1 The problem

The exercise requires to implement a given model for TSP (travelling salesperson problem), considering its practical application: calculate the sequence of holes on a board to be done from a drilling machine such that the total amount of time is minimized.

2 The implementation

The provided file *main.cpp* implements the linear programming model. The structure of the code has been provided by prof. De Giovanni, as well as the *cpxmacro.h* and the *makefile* files. To run the executable file, an instance of the problem has to be provided as argument (example: tsp6.dat).

This is the (simplified) outline of the program:

- Read from file the number of nodes and the weighted adjacency matrix.
- Create the variables (columns in Cplex API) x's and y's. The variable names are stored as *x_i_j* and *y_i_j*.
- Add the constraints (rows in Cplex API).
- Compute the optimal solution and get the time to find it.
- Print the names and values of the variables for the optimal solution. For readability reasons only the variables corresponding the edges that are part of the solution are printed (i.e. variables values different from zero).
- Print the value of the objective function.
- Print the time needed to find the optimal solution. Note that this time considers only the optimization part, not the set up of the problem nor the printing part.

3 The results

The program correctness has been tested manually for graphs of size 4, 5 and 6.
All the following tests have been done in the computers of LabTA in Torre Archimede. The instances of the problem have been created using the program *cdata.cpp*. For each instance, five different version of it have been generated and tested. Each of the shown time results is the average of the five different tests. The code for the automated testing is available in the repository(https://github.com/nicoalbe/tsp-optimization)
Two variables have been considered to make the tests: number of nodes and the shape of the instances. Three shapes for the instances have been used.

Random graph

Given the number of nodes, the program assigns randomly an integer weight to each edge. The tests showed that the range of values have a big impact on the performances, on the same level of the number of nodes. Finding a solution for a graph with random weights between 1 and 100 can take even 10 times more time than finding a solution for a problem where the range is 1 to 10.

It is important to note that these instances are very general and random and don't reflect very well the real problem.

Results Maximum number of nodes to find a solution with a weight range [1,100] in less than:

0,1s : Max $\sim$ 10 nodes.

1s : Max $\sim$ 50 nodes.

10s : Max $\sim$ 80 nodes.

	Time		
Number of nodes	Range [1,10]	Range [1,100]	Range [1,1000]
10	0.052s	0.067s	0.110s
20	0.242s	0.253s	0.33s
50	1.6s	2.89s	2.6s
80	1.98s	15s	22s
100	2.215s	64s	63s
120	5.62s	121s	179s
150	8.36s	341s	448s

Table 1: Results for random graphs.

Random grid graph

The program creates (implicitly) a grid and randomly places the nodes on points of that grid. The weights are calculated as the euclidean distance between two nodes as if they were two points in the Cartesian plane. For example, for the graph in Figure 1, the edge (5,4) has weight 3 and the edge (0,3) has weight $\sqrt{5}$.

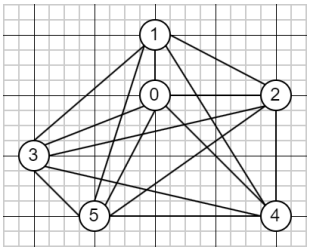

Figure 1: Random grid graph of 6 nodes.

Results The change of dimensions in the grid brings an high change in performances. This brings an high level of uncertainty, so is not possible to provide relevant results about up to which size of the graph the solution is found efficiently. See the different grids for a 50 nodes graph in Table 2 for an example.

Number of nodes	Time
10 (grid 8x10)	0.0711s
20 (grid 8x10)	0.382s
50 (grid 12x10)	4.86s
50 (grid 6x14)	6.47s
50 (grid 100x5)	104s
80 (grid 12x10)	24s
100 (grid 14x18)	131s

Table 2: Results for random grid graphs.

3.1 Realistic graph

This is the case where the shape of the instances is the **closest to the real case** of the boards to be drilled, because of the (assumed) regularity of the position of the holes to be made on the boards.

These instances have been generated in a similar way to the random grid graph, with the only difference that nodes are grouped in sets of four to form rectangles of small but random dimensions. So in this case the grid is (implicitly) generated and then groups of four nodes (in a rectangular disposition) are placed on it.

Results The tests are made on two different instances of the problem with five different versions each (as for all the other tests). In the table the average times are shown along with the single results because of the interesting high time difference between each other. It's important to emphasize how a particular instance of the problem took more than one hour.

Number of nodes	average time	tested times
50 (grid 300x180)	857s	3691s, 35s, 14s, 240s, 304s
80 (grid 400x3000)	149s	391s, 80s, 165s, 57s, 53s

Table 3: Results for random grid graphs.

Conclusion

The tests show poor performances, even on graph of relatively small size. This is unsurprisingly because the tsp is an NP-Hard problem. The c++ program could be made more efficiently (for example, by adding the decision variables all at the same time instead of one at a time) but good enough performances cannot be theoretically reached with our current knowledge of the field.

The tests made on the realistic instances of the problem show poor performances but also high unpredictability on different random versions of the same instance of the problem. The number of tests made is little but good enough to say that this program is not suitable for a practical use in the given real world case, even if we assume the considered instances of the tests as unlucky worst-case examples.